How to Write Well-Behaved Value Wrappers

Simon Brand
@TartanLlama
C++ Developer Advocate, Microsoft
they/them

C++ on Sea
2019-02-05

Please ask questions!

Value Wrappers

Value Wrappers

Types with value-semantics which can store objects of any type

Value Wrappers

Types with value-semantics which can store objects of any type

E.g. `std::pair`, `std::optional`, `std::variant`, `fluent::NamedType`

“Well-Behaved”

Performant

“If a feature is accidentally misapplied by the user and causes what appears to [them] to be an unpredictable result, that feature has a high astonishment factor and is therefore undesirable.”

- Cowlishaw, M. F. (1984). "The design of the REXX language"

Unsurprising

```
/* N.B. GCC has missed optimizations with
Maybe in the past and may generate extra
branches/loads/stores. Use with caution on
hot paths; it's not known whether or not
this is still a problem. */
```

Comparison Operators

```
template<class T>
struct wrapper {
  T t;
};
```

```cpp
template<class T>
struct wrapper {
  T t;
};

wrapper<int> a,b;
a < b;//compiler error
```

~~Unsurprising~~

Comparison Operators

Comparison Operators

```cpp
template <class T, class U>
inline constexpr bool operator==(const optional<T> &lhs,
const optional<U> &rhs) {
  return lhs.has_value() == rhs.has_value() &&
(!lhs.has_value() || *lhs == *rhs);
}
template <class T, class U>
inline constexpr bool operator!=(const optional<T> &lhs,
const optional<U> &rhs) {
  return lhs.has_value() != rhs.has_value() ||
(lhs.has_value() && *lhs != *rhs);
}
template <class T, class U>
inline constexpr bool operator<(const optional<T> &lhs,
const optional<U> &rhs) {
  return rhs.has_value() && (!lhs.has_value() || *lhs < *rhs);
}
template <class T, class U>
inline constexpr bool operator>(const optional<T> &lhs,
const optional<U> &rhs) {
  return lhs.has_value() && (!rhs.has_value() || *lhs > *rhs);
}
template <class T, class U>
inline constexpr bool operator<=(const optional<T> &lhs,
const optional<U> &rhs) {
  return !lhs.has_value() || (rhs.has_value() && *lhs <= *rhs);
}
template <class T, class U>
inline constexpr bool operator>=(const optional<T> &lhs,
const optional<U> &rhs) {
  return !rhs.has_value() || (lhs.has_value() && *lhs >= *rhs);
}
template <class T>
inline constexpr bool operator==(const optional<T> &lhs, nullopt_t) noexcept {
  return !lhs.has_value();
}
template <class T>
inline constexpr bool operator==(nullopt_t, const optional<T> &rhs) noexcept {
  return !rhs.has_value();
}
template <class T>
inline constexpr bool operator!=(const optional<T> &lhs, nullopt_t) noexcept {
  return lhs.has_value();
}
template <class T>
inline constexpr bool operator!=(nullopt_t, const optional<T> &rhs) noexcept {
  return rhs.has_value();
}
```

```cpp
template <class T>
inline constexpr bool operator<(const optional<T> &, nullopt_t) noexcept {
  return false;
}
template <class T>
inline constexpr bool operator<(nullopt_t, const optional<T> &rhs) noexcept {
  return rhs.has_value();
}
template <class T>
inline constexpr bool operator<=(const optional<T> &lhs, nullopt_t) noexcept {
  return !lhs.has_value();
}
template <class T>
inline constexpr bool operator<=(nullopt_t, const optional<T> &) noexcept {
  return true;
}
template <class T>
inline constexpr bool operator>(const optional<T> &lhs, nullopt_t) noexcept {
  return lhs.has_value();
}
template <class T>
inline constexpr bool operator>(nullopt_t, const optional<T> &) noexcept {
  return false;
}
template <class T>
inline constexpr bool operator>=(const optional<T> &, nullopt_t) noexcept {
  return true;
}
template <class T>
inline constexpr bool operator>=(nullopt_t, const optional<T> &rhs) noexcept {
  return !rhs.has_value();
}
template <class T, class U>
inline constexpr bool operator==(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs == rhs : false;
}
template <class T, class U>
inline constexpr bool operator==(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs == *rhs : false;
}
```

```cpp
template <class T, class U>
inline constexpr bool operator!=(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs != rhs : true;
}
template <class T, class U>
inline constexpr bool operator!=(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs != *rhs : true;
}
template <class T, class U>
inline constexpr bool operator<(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs < rhs : true;
}
template <class T, class U>
inline constexpr bool operator<(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs < *rhs : false;
}
template <class T, class U>
inline constexpr bool operator<=(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs <= rhs : true;
}
template <class T, class U>
inline constexpr bool operator<=(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs <= *rhs : false;
}
template <class T, class U>
inline constexpr bool operator>(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs > rhs : false;
}
template <class T, class U>
inline constexpr bool operator>(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs > *rhs : true;
}
template <class T, class U>
inline constexpr bool operator>=(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs >= rhs : false;
}
template <class T, class U>
inline constexpr bool operator>=(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs >= *rhs : true;
}
```

Comparison Operators

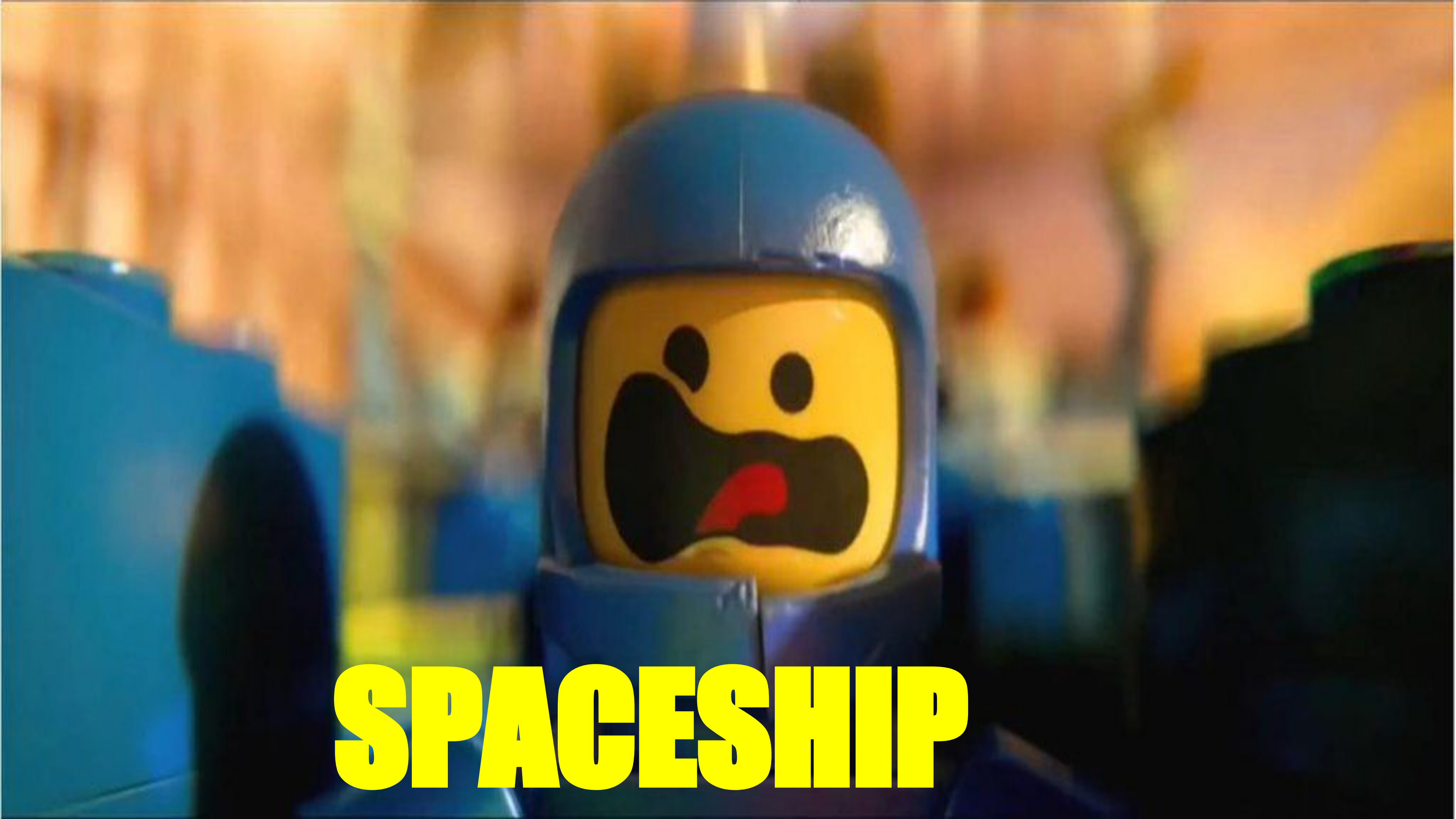
```
wrapper<int> a,b;
a < b; //compiles
```

Unsurprising

Unsurprising

(But a ton of code)


THE LEGO® MOVIE


SPACESHIP

<=>

The expression returns an object such that

- `(a <=> b) < 0` if `a < b`
- `(a <=> b) > 0` if `a > b`
- `(a <=> b) == 0` if a and b are equal/equivalent.

cppreference.com

```cpp
template<class T, class U>
struct pair {
  T t;
  U u;
};
```

```
auto operator<=>(pair const& rhs) const
  = default;
```

```cpp
auto operator<=> (pair const& rhs) const
-> /* a mess, because C++ */
{
  if (auto cmp = t<=>rhs.t; cmp != 0)
    return cmp;
  return u <=> rhs.u;
}
```

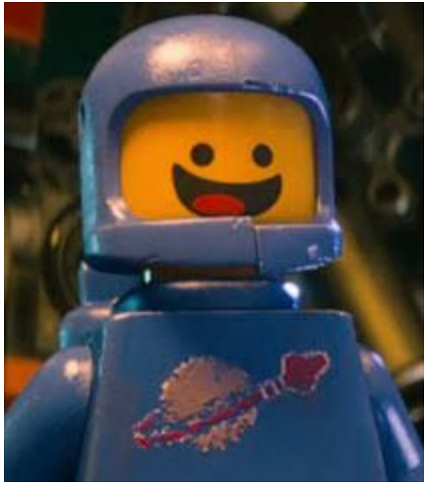
Comparison Operators

```
template <class T, class U>
inline constexpr bool operator==(const optional<T> &lhs,
const optional<U> &rhs) {
  return lhs.has_value() == rhs.has_value() &&
(!lhs.has_value() || *lhs == *rhs);
}
template <class T, class U>
inline constexpr bool operator!=(const optional<T> &lhs,
const optional<U> &rhs) {
  return lhs.has_value() != rhs.has_value() ||
(lhs.has_value() && *lhs != *rhs);
}
template <class T, class U>
inline constexpr bool operator<(const optional<T> &lhs,
const optional<U> &rhs) {
  return rhs.has_value() && (!lhs.has_value() || *lhs < *rhs);
}
template <class T, class U>
inline constexpr bool operator>(const optional<T> &lhs,
const optional<U> &rhs) {
  return lhs.has_value() && (!rhs.has_value() || *lhs > *rhs);
}
template <class T, class U>
inline constexpr bool operator<=(const optional<T> &lhs,
const optional<U> &rhs) {
  return !lhs.has_value() || (rhs.has_value() && *lhs <= *rhs);
}
template <class T, class U>
inline constexpr bool operator>=(const optional<T> &lhs,
const optional<U> &rhs) {
  return !rhs.has_value() || (lhs.has_value() && *lhs >= *rhs);
}
template <class T>
inline constexpr bool operator==(const optional<T> &lhs, nullopt_t) noexcept {
  return !lhs.has_value();
}
template <class T>
inline constexpr bool operator==(nullopt_t, const optional<T> &rhs) noexcept {
  return !rhs.has_value();
}
template <class T>
inline constexpr bool operator!=(const optional<T> &lhs, nullopt_t) noexcept {
  return lhs.has_value();
}
template <class T>
inline constexpr bool operator!=(nullopt_t, const optional<T> &rhs) noexcept {
  return rhs.has_value();
}
```

```
template <class T>
inline constexpr bool operator<(const optional<T> &, nullopt_t) noexcept {
  return false;
}
template <class T>
inline constexpr bool operator<(nullopt_t, const optional<T> &rhs) noexcept {
  return rhs.has_value();
}
template <class T>
inline constexpr bool operator<=(const optional<T> &lhs, nullopt_t) noexcept {
  return !lhs.has_value();
}
template <class T>
inline constexpr bool operator<=(nullopt_t, const optional<T> &) noexcept {
  return true;
}
template <class T>
inline constexpr bool operator>(const optional<T> &lhs, nullopt_t) noexcept {
  return lhs.has_value();
}
template <class T>
inline constexpr bool operator>(nullopt_t, const optional<T> &) noexcept {
  return false;
}
template <class T>
inline constexpr bool operator>=(const optional<T> &, nullopt_t) noexcept {
  return true;
}
template <class T>
inline constexpr bool operator>=(nullopt_t, const optional<T> &rhs) noexcept {
  return !rhs.has_value();
}
template <class T, class U>
inline constexpr bool operator==(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs == rhs : false;
}
template <class T, class U>
inline constexpr bool operator==(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs == *rhs : false;
}
```

```
template <class T, class U>
inline constexpr bool operator!=(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs != rhs : true;
}
template <class T, class U>
inline constexpr bool operator!=(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs != *rhs : true;
}
template <class T, class U>
inline constexpr bool operator<(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs < rhs : true;
}
template <class T, class U>
inline constexpr bool operator<(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs < *rhs : false;
}
template <class T, class U>
inline constexpr bool operator<=(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs <= rhs : true;
}
template <class T, class U>
inline constexpr bool operator<=(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs <= *rhs : false;
}
template <class T, class U>
inline constexpr bool operator>(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs > rhs : false;
}
template <class T, class U>
inline constexpr bool operator>(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs > *rhs : true;
}
template <class T, class U>
inline constexpr bool operator>=(const optional<T> &lhs, const U &rhs) {
  return lhs.has_value() ? *lhs >= rhs : false;
}
template <class T, class U>
inline constexpr bool operator>=(const U &lhs, const optional<T> &rhs) {
  return rhs.has_value() ? lhs >= *rhs : true;
}
```

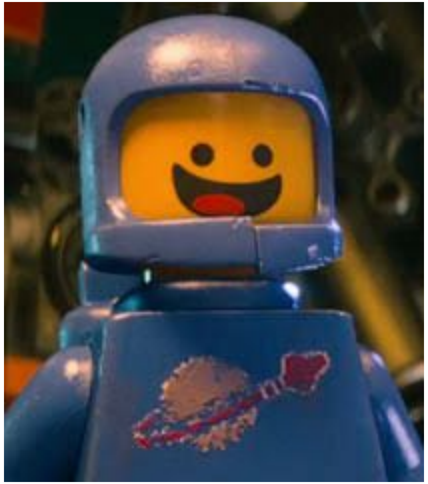
Spaceships!



```
template <typename U>
constexpr auto operator<=>(optional<U> const& rhs)
const
  -> decltype(compare_3way(**this, *rhs))
{
  if (has_value() && rhs.has_value()) {
    return compare_3way(**this, *rhs);
  } else {
    return has_value() <=> rhs.has_value();
  }
}

template <typename U>
constexpr auto operator<=>(U const& rhs) const
  -> decltype(compare_3way(**this, rhs))
{
  if (has_value()) {
    return compare_3way(**this, rhs);
  } else {
    return strong_ordering::less;
  }
}
```

```
constexpr strong_ordering
operator<=>(nullopt_t ) const
{
  return has_value() ?
              strong_ordering::greater :
              strong_ordering::equal;
}
```



Lovingly stolen from Barry Revzin

Unsurprising

(But a ton of code)

Unsurprising

noexcept Propagation

Propagate the noexcept-ness of move and swap operations

Propagating noexcept

```
std::vector<optional<tracer>> v;
std::fill_n(std::back_inserter(v), 100000, tracer{});
```

Propagating noexcept

```
std::vector<optional<tracer>> v;
std::fill_n(std::back_inserter(v), 100000, tracer{});
```

Without propagation	With propagation
231,071 copies, 100,000 moves	100,000 copies, 231,071 moves

~~Performant~~

Propagating noexcept

```
optional(optional &&rhs)
                                                  {
  if (rhs.has_value()) {
    this->construct(std::move(rhs.get()));
  } else {
    this->m_has_value = false;
  }
}
```

Propagating noexcept

```
optional(optional &&rhs) noexcept(
 std::is_nothrow_move_constructible<T>::value) {
  if (rhs.has_value()) {
    this->construct(std::move(rhs.get()));
  } else {
    this->m_has_value = false;
  }
}
```

Performant

To explicit, or not to explicit, that is the question.

- Hamlet++,
  William Fakespeare

To explicit or not to explicit

```cpp
template<class T>
struct wrapper {
  T t;
};
```

```cpp
template<class T>
struct wrapper {
  T t;
};

//compiler error
wrapper<int> wi = 0;
```

```
wrapper (T const& t) :
    t(t) {}
```

```cpp
template <class U>
wrapper (U&& u) :
   t(std::forward<U>(u)) {}
```

```
struct oh_no {
  explicit oh_no(bool);
};
```

```
struct oh_no {
  explicit oh_no(bool);
};

void do_thing(wrapper<oh_no>);
```

```cpp
struct oh_no {
  explicit oh_no(bool);
};

void do_thing(wrapper<oh_no>);

//compiles, but shouldn't
do_thing(true);
```

~~Unsurprising~~

```
template<class U, std::enable_if_t<
              std::is_constructible<T, U>::value &&
               std::is_convertible<U, T>::value
            >* = nullptr
           >
/* not explicit */
wrapper(U&& u) : t(std::forward<U>(u))
{}
```

```
template<class U, std::enable_if_t<
              std::is_constructible<T, U>::value &&
              !std::is_convertible<U, T>::value
            >* = nullptr
          >
explicit
wrapper(U&& u) : t(std::forward<U>(u))
{}
```

```cpp
template<class T>
struct wrapper {
  T t;
};

//compiles
wrapper<int> wi = 0;
```

```cpp
struct oh_no {
  explicit oh_no(bool);
};

void do_thing(wrapper<oh_no>);

//doesn’t compile, as expected
do_thing(true);
```

Unsurprising

Unsurprising

(With code duplication)

`noexcept(bool)`

```
noexcept(bool)
explicit(bool)?
```

```
template<class U, std::enable_if_t<
            std::is_constructible<T, U>::value &&
             std::is_convertible<U, T>::value
          >* = nullptr
         >

wrapper(U&& u) : t(std::forward<U>(u))
{}
```

```
template<class U, std::enable_if_t<
            std::is_constructible<T, U>::value &&
            !std::is_convertible<U, T>::value
          >* = nullptr
        >
explicit
wrapper(U&& u) : t(std::forward<U>(u))
{}
```

```cpp
template<class U, std::enable_if_t<
            std::is_constructible<T, U>::value

          >* = nullptr
         >
explicit(!std::is_convertible<U, T>::value)
wrapper(U&& u) : t(std::forward<U>(u))
{}
```

Unsurprising

(With code duplication)

Unsurprising

Conditionally Deleting Special Members


ANGLIA, SCOTIA et HIBERNIA.
OCEANVS DEVCALI
OCEANVS
DONIVS
Hiberniæ aliquot nomina
OCEANVS
Milliaria Anglica parua
VERGIVIVS OCEANVS
Per Gerardum Mercatorem Cum Privilegio
HIBERNI
MARE
OCEANVS BRITANNICVS
Scotiæ
Angliæ


OCEANVS DEVCALI
ANGLIA, SCOTIA et HIBERNIA.
DONIVS
OCEANVS
Hiberniæ aliquot nomina
OCEANVS
Milliaria Anglica parua
VERGIVIVS OCEANVS
HIBERNI
MARE
OCEANVS BRITANNICVS
DANIAE PARS
Per Gerardum Mercatorem Cum Privilegio

```cpp
template <class T>
struct optional {
  union {
    T t;
    char dummy;
  };
  bool engaged = false;
};
```

```cpp
optional(optional const& rhs) {
  if (rhs.engaged) {
    new (std::addressof(t)) T (rhs.t);
    engaged = true;
  }
}
```

```
~optional() {
  if (engaged) t.~T();
}
```

Conditionally Deleting Special Members

```
optional<std::unique_ptr<int>> a;
optional<std::unique_ptr<int>> b = a;
```

Conditionally Deleting Special Members

```
optional<std::unique_ptr<int>> a;
optional<std::unique_ptr<int>> b = a;
```

Without deletion	With deletion
`<source>:356:41: error: call to deleted constructor of 'std::unique_ptr<int, std::default_delete<int> >'` `    new (std::addressof(this->m_value)) T(std::forward<Args>(args)...);`	`<source>:2299:40: error: call to implicitly-deleted copy constructor of 'tl::optional<std::unique_ptr<int> >'` `    tl::optional<std::unique_ptr<int>> b = a;`

Conditionally Deleting Special Members

optional<std::unique_ptr<int>> a;
optional<std::unique_ptr<int>> b = a;

Without deletion	With deletion
`<source>:356:41: error: call to deleted constructor of 'std::unique_ptr<int, std::default_delete<int> >'` `new (std::addressof(this->m_value)) T(std::forward<Args>(args)...);`	`<source>:2299:40: error: call to implicitly-deleted copy constructor of 'tl::optional<std::unique_ptr<int> >'` `tl::optional<std::unique_ptr<int>> b = a;`

Conditionally Deleting Special Members

```
std::is_copy_constructible<
    optional<std::unique_ptr<int>>
>::value
```

Conditionally Deleting Special Members

```
std::is_copy_constructible<
    optional<std::unique_ptr<int>>
>::value
```

Without deletion	With deletion
`true`	`false`

~~Unsurprising~~

```cpp
template <class T>
struct optional       {
  union {
    T t;
    char dummy;
  };
  bool engaged = false;
  //ctors, dtors
};
```

```
template <class T>
struct optional_base {
  union {
    T t;
    char dummy;
  };
  bool engaged = false;
  //ctors, dtors
};
```

```cpp
template <class T>
struct optional : private optional_base<T>,
                  private delete_ctor_base<T>,
                  private delete_assign_base<T> {
  //other member functions
};
```

```cpp
template <class T,
          bool EnableCopy =
            std::is_copy_constructible<T>::value,
          bool EnableMove =
            std::is_move_constructible<T>::value>
struct delete_ctor_base;
```

Conditionally Deleting Special Members

```
template <class T>
struct delete_ctor_base<true, true> {
  delete_ctor_base() = default;

  delete_ctor_base(const delete_ctor_base &) = default;
  delete_ctor_base(delete_ctor_base &&) noexcept = default;

  delete_ctor_base &
  operator=(const delete_ctor_base &) = default;

  delete_ctor_base &
  operator=(optional_delete_ctor_base &&) noexcept = default;
};
```

Conditionally Deleting Special Members

```
template <class T>
struct delete_ctor_base<true, false> {
  delete_ctor_base() = default;

  delete_ctor_base(const delete_ctor_base &) = default;
  delete_ctor_base(delete_ctor_base &&) noexcept = delete;

  delete_ctor_base &
  operator=(const delete_ctor_base &) = default;

  delete_ctor_base &
  operator=(optional_delete_ctor_base &&) noexcept = default;
};
```

Conditionally Deleting Special Members

```
template <class T>
struct delete_ctor_base<false, true> {
  delete_ctor_base() = default;

  delete_ctor_base(const delete_ctor_base &) = delete;
  delete_ctor_base(delete_ctor_base &&) noexcept = default;

  delete_ctor_base &
  operator=(const delete_ctor_base &) = default;

  delete_ctor_base &
  operator=(optional_delete_ctor_base &&) noexcept = default;
};
```

Conditionally Deleting Special Members

```
template <class T>
struct delete_ctor_base<false, false> {
  delete_ctor_base() = default;

  delete_ctor_base(const delete_ctor_base &) = delete;
  delete_ctor_base(delete_ctor_base &&) noexcept = delete;

  delete_ctor_base &
  operator=(const delete_ctor_base &) = default;

  delete_ctor_base &
  operator=(optional_delete_ctor_base &&) noexcept = default;
};
```

Conditionally Deleting Special Members

```
template <class T,

          bool EnableCopy =
        (std::is_copy_constructible<T>::value &&
         std::is_copy_assignable<T>::value),

          bool EnableMove =
        (std::is_move_constructible<T>::value &&
         std::is_move_assignable<T>::value)>

struct delete_assign_base;
```

Conditionally Deleting Special Members

```
//compiles
static_assert(
 !std::is_copy_constructible<
    optional<std::unique_ptr<int>>
  >::value
);
```

Unsurprising

Unsurprising

(With a ton of code)

Concepts

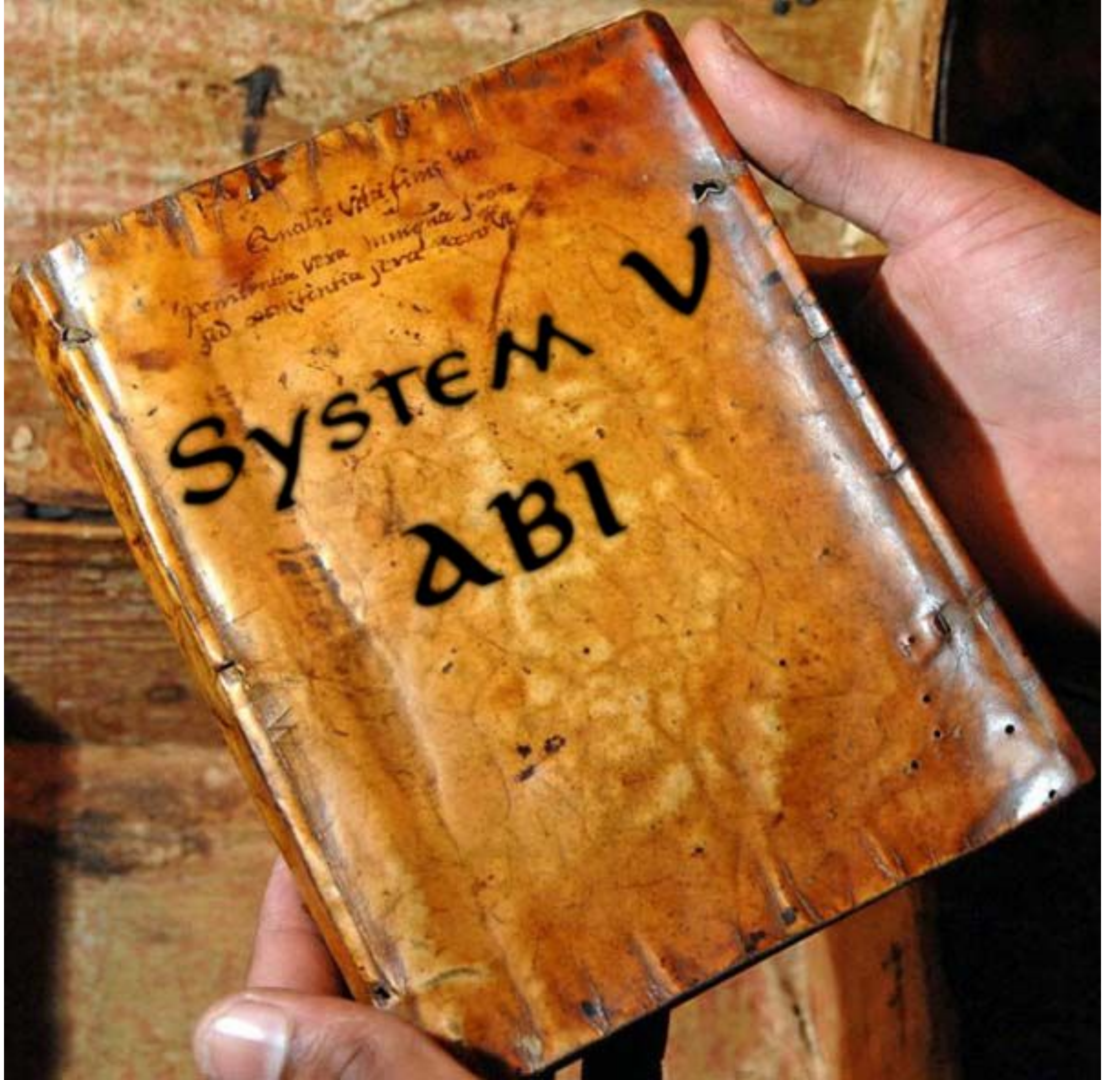
```
template <class T>
struct optional {
  optional(optional const&)
    requires is_copy_constructible_v<T>
  { /*...*/ }
  optional(optional&&)
    requires is_move_constructible_v<T>
  { /*...*/ }
};
```

Unsurprising

(With a ton of code)

Unsurprising

Triviality Propagation


System V
ABI

“An object with either a non-trivial copy constructor or a non-trivial destructor cannot be passed by value because such objects must have well defined addresses.”

Propagating Trivial-Copyability

Trivial copy-constructor:

- Defaulted or implicitly-defined
- No virtual member functions or base classes
- All base and member copy ctors are trivial

Propagating Trivial-Destructibility

Trivial destructor:

- Defaulted or implicitly-defined
- Not virtual
- All base and member destructors are trivial

```
static_assert(
 std::is_trivially_copy_constructible<
    optional<int>
  >::value
);
```

Triviality Propagation

```
//compiler error
static_assert(
 std::is_trivially_copy_constructible<
     optional<int>
   >::value
);
```

Triviality Propagation

```
//compiler error
static_assert(
 std::is_trivially_destructible<
    optional<int>
  >::value
);
```

~~Performant~~

```
optional<int> get_42() { return 42; }
```

Propagating Trivial Constructors

```
optional<int> get_42() { return 42; }
```

Without propagation	With propagation
get_42(): # @get_42() mov dword ptr [rdi], 42 mov byte ptr [rdi + 4], 1 mov rax, rdi ret	get_42(): # @get_42() movabs rax, 4294967338 ret

Propagating Trivial Constructors

```
void take_opt(optional<int>);
void call() { take_opt(42); }
```

Without propagation	With propagation
call(): # @call() push rax mov dword ptr [rsp], 42 mov byte ptr [rsp + 4], 1 mov rdi, rsp call take_opt(optional<int>) pop rax ret	call(): # @call() movabs rdi, 4294967338 jmp take_opt(optional<int>)

Propagating Trivial-Destructability

```
void take_opt(optional<int>);
void call() { take_opt(42); }
```

<table>
<tr><th>Without propagation</th><th>With propagation</th></tr>
<tr><td><pre>call(): # @call()
push rax
mov dword ptr [rsp], 42
mov byte ptr [rsp + 4], 1
mov rdi, rsp
call take_opt(optional<int>)
cmp byte ptr [rsp + 4], 0
je .LBB0_3
mov byte ptr [rsp + 4], 0
.LBB0_3:
pop rax
ret
cmp byte ptr [rsp + 4], 0
je .LBB0_6
mov byte ptr [rsp + 4], 0
.LBB0_6:
mov rdi, rax
call _Unwind_Resume</pre></td><td><pre>call(): # @call()
movabs rdi, 4294967338
jmp take_opt(optional<int>)</pre></td></tr>
</table>

Propagating Trivial-Destructability

```
std::vector<tl::optional<int>> a();
void b() { a(); }
```

Without propagation	With propagation

b(): # @b()
sub rsp, 24
mov rdi, rsp
call a()
mov rdi, qword ptr [rsp]
mov rax, qword ptr [rsp + 8]
cmp rdi, rax
je .LBB0_16
lea rdx, [rax - 8]
sub rdx, rdi
mov esi, edx
shr esi, 3
add esi, 1
mov rcx, rdi
and rsi, 3
je .LBB0_6
neg rsi
mov rcx, rdi
.LBB0_3:
cmp byte ptr [rcx + 4], 0
je .LBB0_5
mov byte ptr [rcx + 4], 0
.LBB0_5:
add rcx, 8
add rsi, 1
jne .LBB0_3
.LBB0_6:
cmp rdx, 24
jb .LBB0_16
.LBB0_7:
cmp byte ptr [rcx + 4], 0
je .LBB0_9
mov byte ptr [rcx + 4], 0
.LBB0_9:
cmp byte ptr [rcx + 12], 0
je .LBB0_11
mov byte ptr [rcx + 12], 0
.LBB0_11:
cmp byte ptr [rcx + 20], 0
je .LBB0_13
mov byte ptr [rcx + 20], 0
.LBB0_13:
cmp byte ptr [rcx + 28], 0
je .LBB0_15
mov byte ptr [rcx + 28], 0
.LBB0_15:
add rcx, 32
cmp rax, rcx
jne .LBB0_7
.LBB0_16:
test rdi, rdi
je .LBB0_18
call operator delete(void*)
.LBB0_18:
add rsp, 24
ret
	b(): # @b()
	sub rsp, 24
	mov rdi, rsp
	call a()
	mov rdi, qword ptr [rsp]
	test rdi, rdi
	je .LBB0_2
	call operator delete(void*)
	.LBB0_2:
	add rsp, 24
	ret

```
~optional_base() {
  if (engaged) t.~T();
}
```

```cpp
template<
  class U=T,
  std::enable_if_t<
    std::is_trivially_destructible<U>::value>* =
      nullptr>
~optional_base() {
  if (engaged) t.~T();
}
```

```cpp
template<
  class U=T,
  std::enable_if_t<
    std::is_trivially_destructible<U>::value>* =
      nullptr>
~optional_base() {
  if (engaged) t.~T();
}
```

```
template <class T, bool =
          std::is_trivially_destructible<T>::value>
struct optional_storage_base {
  //<union and `engaged`>

  ~optional_storage_base() {
    if (engaged) t.~T();
  }
};
```

Triviality Propagation

```
template <class T>

struct optional_storage_base<T,true> {
  //<union and `engaged`>

  ~optional_storage_base() = default;


};
```

```cpp
template <class T, bool = std::is_trivially_copy_constructible<T>::value>
struct optional_copy_base : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;
};

template <class T>
struct optional_copy_base<T, false> : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;

  optional_copy_base() = default;
  optional_copy_base(const optional_copy_base &rhs) {
    if (rhs.has_value()) {
      new (this->t) T(rhs.t);
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_copy_base(optional_copy_base &&rhs) = default;
  optional_copy_base &operator=(const optional_copy_base &rhs) = default;
  optional_copy_base &operator=(optional_copy_base &&rhs) = default;
}
```

```cpp
template <class T, bool = std::is_trivially_copy_constructible<T>::value>
struct optional_copy_base : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;
};

template <class T>
struct optional_copy_base<T, false> : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;

  optional_copy_base() = default;
  optional_copy_base(const optional_copy_base &rhs) {
    if (rhs.has_value()) {
      new (this->t) T(rhs.t);
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_copy_base(optional_copy_base &&rhs) = default;
  optional_copy_base &operator=(const optional_copy_base &rhs) = default;
  optional_copy_base &operator=(optional_copy_base &&rhs) = default;
}

template <class T, bool = std::is_trivially_move_constructible<T>::value>
  struct optional_move_base : optional_copy_base<T> {
  using optional_copy_base<T>::optional_copy_base;
};

template <class T>
struct optional_move_base<T, false> : optional_copy_base<T> {
  using optional_copy_base<T>::optional_copy_base;

  optional_move_base() = default;
  optional_move_base(const optional_move_base &rhs) = default;
  optional_move_base(optional_move_base &&rhs) {
    if (rhs.has_value()) {
      new (this->t) T(std::move(rhs.t));
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_move_base &operator=(const optional_move_base &rhs) = default;
  optional_move_base &operator=(optional_move_base &&rhs) = default;
};
```

```
template <class T, bool = std::is_trivially_copy_constructible<T>::value>
struct optional_copy_base : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;
};

template <class T>
struct optional_copy_base<T, false> : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;

  optional_copy_base() = default;
  optional_copy_base(const optional_copy_base &rhs) {
    if (rhs.has_value()) {
      new (this->t) T(rhs.t);
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_copy_base(optional_copy_base &&rhs) = default;
  optional_copy_base &operator=(const optional_copy_base &rhs) = default;
  optional_copy_base &operator=(optional_copy_base &&rhs) = default;
}

template <class T, bool = std::is_trivially_move_constructible<T>::value>
  struct optional_move_base : optional_copy_base<T> {
  using optional_copy_base<T>::optional_copy_base;
};

template <class T>
struct optional_move_base<T, false> : optional_copy_base<T> {
  using optional_copy_base<T>::optional_copy_base;

  optional_move_base() = default;
  optional_move_base(const optional_move_base &rhs) = default;
  optional_move_base(optional_move_base &&rhs) {
    if (rhs.has_value()) {
      new (this->t) T(std::move(rhs.t));
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_move_base &operator=(const optional_move_base &rhs) = default;
  optional_move_base &operator=(optional_move_base &&rhs) = default;
};
```

```
template <class T, bool = std::is_trivially_copy_assignable<T>::value>
struct optional_copy_assign_base : optional_move_base<T> {
  using optional_move_base<T>::optional_move_base;
};
template <class T>
struct optional_copy_assign_base<T, false> : optional_move_base<T> {
  using optional_move_base<T>::optional_move_base;
  optional_copy_assign_base() = default;
  optional_copy_assign_base(const optional_copy_assign_base &rhs) = default;
  optional_copy_assign_base(optional_copy_assign_base &&rhs) = default;
  optional_copy_assign_base &operator=(const optional_copy_assign_base &rhs) {
    if (this->enabled) {
      if (rhs.enabled) this->t = rhs.t;
      else {
        this->t.~T(); this->enabled = false;
      }
    }
    if (rhs.enabled) {
      new (this->t) T (rhs.t); this->enabled = true;
    }
  }
  optional_copy_assign_base &operator=(optional_copy_assign_base &&rhs) = default;
};
```

```cpp
template <class T, bool = std::is_trivially_copy_constructible<T>::value>
struct optional_copy_base : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;
};

template <class T>
struct optional_copy_base<T, false> : optional_storage_base<T> {
  using optional_storage_base<T>::optional_storage_base;

  optional_copy_base() = default;
  optional_copy_base(const optional_copy_base &rhs) {
    if (rhs.has_value()) {
      new (this->t) T(rhs.t);
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_copy_base(optional_copy_base &&rhs) = default;
  optional_copy_base &operator=(const optional_copy_base &rhs) = default;
  optional_copy_base &operator=(optional_copy_base &&rhs) = default;
}

template <class T, bool = std::is_trivially_move_constructible<T>::value>
  struct optional_move_base : optional_copy_base<T> {
  using optional_copy_base<T>::optional_copy_base;
};

template <class T>
struct optional_move_base<T, false> : optional_copy_base<T> {
  using optional_copy_base<T>::optional_copy_base;

  optional_move_base() = default;
  optional_move_base(const optional_move_base &rhs) = default;
  optional_move_base(optional_move_base &&rhs) {
    if (rhs.has_value()) {
      new (this->t) T(std::move(rhs.t));
      this->enabled = true;
    } else {
      this->enabled = false;
    }
  }
  optional_move_base &operator=(const optional_move_base &rhs) = default;
  optional_move_base &operator=(optional_move_base &&rhs) = default;
};
```

```cpp
template <class T, bool = std::is_trivially_copy_assignable<T>::value>
struct optional_copy_assign_base : optional_move_base<T> {
  using optional_move_base<T>::optional_move_base;
};
template <class T>
struct optional_copy_assign_base<T, false> : optional_move_base<T> {
  using optional_move_base<T>::optional_move_base;
  optional_copy_assign_base() = default;
  optional_copy_assign_base(const optional_copy_assign_base &rhs) = default;
  optional_copy_assign_base(optional_copy_assign_base &&rhs) = default;
  optional_copy_assign_base &operator=(const optional_copy_assign_base &rhs) {
    if (this->enabled) {
      if (rhs.enabled) this->t = rhs.t;
      else {
        this->t.~T(); this->enabled = false;
      }
    }
    if (rhs.enabled) {
      new (this->t) T (rhs.t); this->enabled = true;
    }
  }
  optional_copy_assign_base &operator=(optional_copy_assign_base &&rhs) = default;
};
template <class T, bool = std::is_trivially_move_assignable<T>::value>
struct optional_move_assign_base : optional_copy_assign_base<T> {
  using optional_copy_assign_base<T>::optional_copy_assign_base;
};
template <class T>
struct optional_move_assign_base<T, false> : optional_copy_assign_base<T> {
  using optional_copy_assign_base<T>::optional_copy_assign_base;
  optional_move_assign_base() = default;
  optional_move_assign_base(const optional_move_assign_base &rhs) = default;
  optional_move_assign_base(optional_move_assign_base &&rhs) = default;
  optional_move_assign_base &operator=(const optional_move_assign_base &rhs) = default;
  optional_move_assign_base &operator=(optional_move_assign_base &&rhs) {
    if (this->enabled) {
      if (rhs.enabled) this->t = std::move(rhs.t);
      else {
        this->t.~T(); this->enabled = false;
      }
    }
    if (rhs.enabled) {
      new (this->t) T (std::move(rhs.t); this->enabled = true;
    }
  }
};
```

```cpp
template <class T>
struct optional
  : private optional_move_assign_base<T> {
  //...
  ~optional() = default;
};
```

```
//compiles
static_assert(
 std::is_trivially_copy_constructible<
    optional<int>
  >::value
);
```

```
//compiles
static_assert(
 std::is_trivially_destructible<
    optional<int>
  >::value
);
```

Performant

Performant

(But a ton of code)

Concepts + P0848

```
template <class T>
struct optional {
  optional (optional const& rhs) requires
    is_trivially_copy_constructible_v<T> &&
    is_copy_constructible_v<T> = default;

  optional(optional const& rhs) requires
    is_copy_constructible_v<_Tp>
  { /*...*/ }
};
```

Performant

(But a ton of code)

Performant

JF Bastien
@jfbastien

Q: How do you know if a C++ developer is qualified?

6:13 PM - Jul 13, 2018

217 61 people are talking about this

JF Bastien
@jfbastien

Q: How do you know if a C++ developer is qualified?
A: you look at their CV.

6:13 PM - Jul 13, 2018

♡ 217 💬 61 people are talking about this

JF Bastien
@jfbastien

Q: How do you know if a C++ developer is qualified?
A: you look at their references.

6:13 PM - Jul 13, 2018

217 61 people are talking about this

Ref-Qualified Accessor Functions

Ref-qualified Accessor Functions

```
T& operator*() /*nothing*/
{ return this->m_value; }

T const& operator*() const
{ return this->m_value; }
```

Ref-qualified Accessor Functions

```
auto x = *std::move(opt);
auto y = std::move(*opt);
```

Ref-qualified Accessor Functions

```
auto x = *std::move(opt); //copies
auto y = std::move(*opt); //moves
```

~~Performant~~

~~Unsurprising~~

Ref-qualified Accessor Functions

```
T& operator*() /*nothing*/
{ return this->m_value; }

T const& operator*() const
{ return this->m_value; }
```

Ref-qualified Accessor Functions

```
T& operator*() &
{ return this->m_value; }

T const& operator*() const &
{ return this->m_value; }

T&& operator*() &&
{ return std::move(this->m_value); }

T const&& operator*() const &&
{ return std::move(this->m_value); }
```

`const&& !?!?`

```
auto x = std::cref(*get_const_optional());
```

const&& !?!?

auto x = std::cref(*get_const_optional());

Without const&&	With const&&

`Delete hard drive, order pizza, Robin Williams fills your living room with Rhinos`	`<source>:23:5: error: call to deleted function 'cref'`
	`std::cref(*get_const_optional());`
	`^~~~~~~~~`

Ref-qualified Accessor Functions

```
T& operator*() &
{ return this->m_value; }

T const& operator*() const &
{ return this->m_value; }

T&& operator*() &&
{ return std::move(this->m_value); }

T const&& operator*() const &&
{ return std::move(this->m_value); }
```

Performant
Unsurprising

Performant
Unsurprising

(But with code duplication)

Deducing this

```
template <class Self>
decltype(auto) operator*(this Self&& self) {
  return std::forward<Self>(self).m_value;
}
```

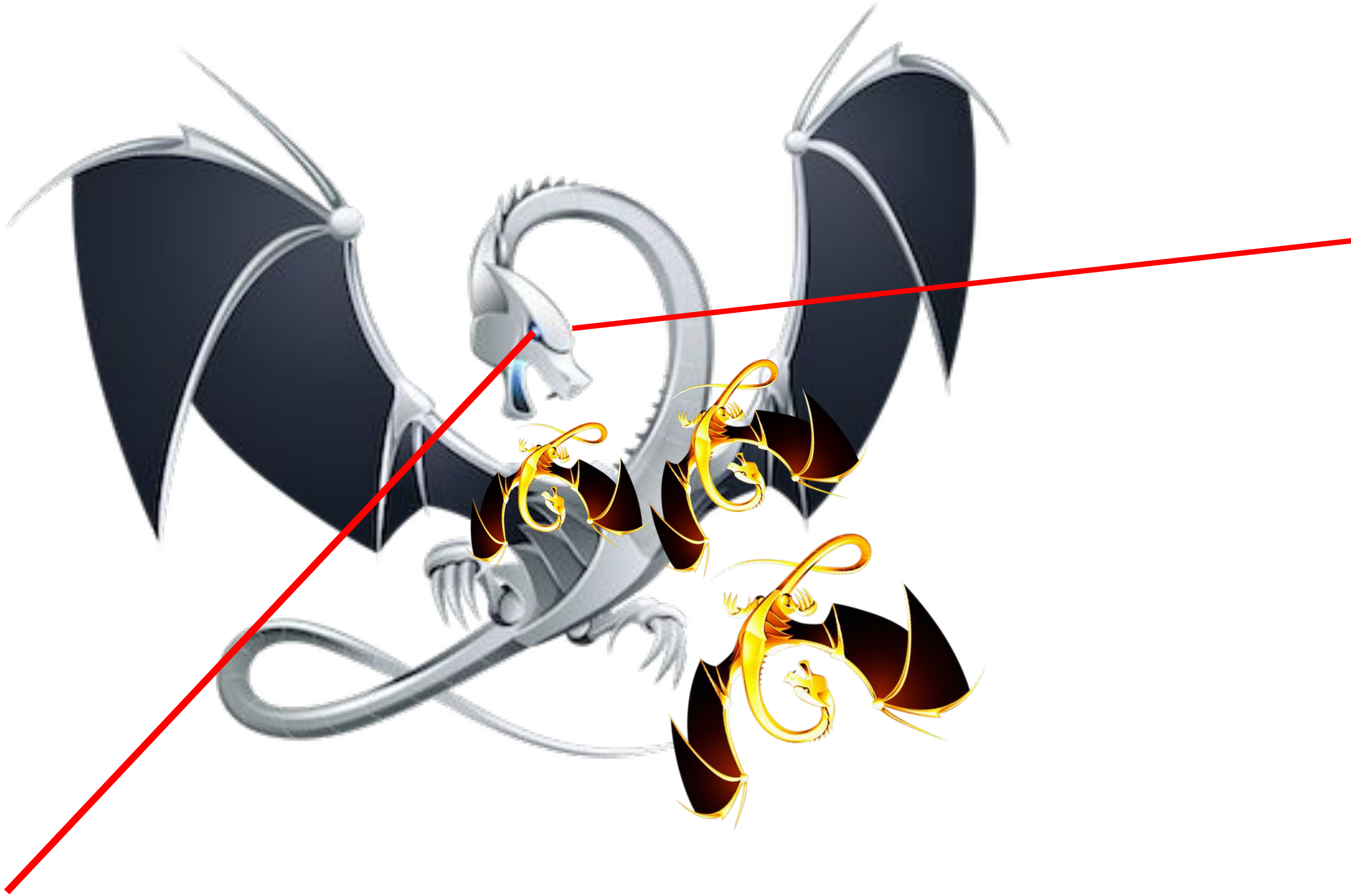
Performant
Unsurprising

(But with code duplication)

Performant
Unsurprising

SFINAE-Unfriendly Callables

ANGLIA, SCOTIA et HIBERNIA.
OCEANVS DEVCALI:
OCEANVS
DONIVS
DANIAE PARS
OCCIDEN
Hiberniæ aliquot nomina
OCEANVS
HIBERNI
MARE
Milliaria Anglica parua
VERGIVIVS OCEANVS
OCEANVS BRITANNICVS



```cpp
template<class T>
struct wrapper {
  T t;
};
```

```cpp
template<class T>
struct wrapper {
    T t;
    template<class F>
    auto pass_to (F f) /* */ -> decltype(f(t)) {
        return f(t);
    }
    template<class F>
    auto pass_to (F f) const -> decltype(f(t)) {
        return f(t);
    }
};
```

```cpp
struct foo {
  void do_thing();
};

wrapper<foo> f;
f.pass_to([](auto&& x){x.do_thing();});
```

```
<source>:22:26: error: passing 'const foo' as 'this' argument discards qualifiers [-fpermissive]
  f.pass_to([](auto&& x){x.do_thing();});
                         ^
<source>:17:10: note:   in call to 'void foo::do_thing()'
    void do_thing();
```

~~Unsurprising~~

```cpp
template<class T>
struct wrapper {
    T t;
    template<class F>
    auto pass_to (F f) /* */ -> decltype(f(t)) {
        return f(t);
    }
    template<class F>
    auto pass_to (F f) const -> decltype(f(t)) {
        return f(t);
    }
};
```

```
template<class T>
struct wrapper {
    T t;
    template<class F>
    auto pass_to (F f) /* */            {
        return f(t);
    }
    template<class F>
    auto pass_to (F f) const            {
        return f(t);
    }
};
```

```cpp
struct foo {
  void do_thing();
};

wrapper<foo> f;
//compiles
f.pass_to([](auto&& x){x.do_thing();});
```

Unsurprising

Unsurprising

(But no longer SFINAE-friendly)

```cpp
template<class T>
struct wrapper {
    T t;
    template<class F>
    auto pass_to (F f) /* */ -> decltype(f(t)) {
        return f(t);
    }
    template<class F>
    auto pass_to (F f) const -> decltype(f(t)) {
        return f(t);
    }
};
```

```
template<class T>
struct wrapper {
    T t;
    template<class Self, class F>
    auto pass_to (this Self&& self, F f) ->
          decltype(f(self.t)) {
        return f(self.t);
    }



};
};
```

Unsurprising

(But no longer SFINAE-friendly)

Unsurprising

```
/* N.B. GCC has missed optimizations with
Maybe in the past and may generate extra
branches/loads/stores. Use with caution on
hot paths; it's not known whether or not
this is still a problem. */
```

Topics covered

- Comparison operators
- Noexcept propagation
- Conditional explicitness
- Conditionally deleting special members
- Triviallity propagation
- Ref-qualified member accessors
- SFINAE-unfriendly callables

How to Write Well-Behaved Value Wrappers

Simon Brand
@TartanLlama
C++ Developer Advocate, Microsoft
they/them

C++ on Sea
2019-02-05

Resources

https://goo.gl/aFuiA9

Talk to me about:

- Visual Studio
- Visual Studio Code
- Vcpkg
- Weird films